PostgreSQL 请求优化机制研究

刘长浩　孙玉芳
（中科院软件所　北京100080）

摘　要　作为功能最强大、特性最丰富和最复杂的自由软件数据库管理系统，PostgreSQL 不仅获得了数据库研究人员的青睐，而且被越来越多地应用到商业中来，显示了非常好的发展前景。本文讨论了 PostgreSQL 数据库管理系统请求优化模块的实现，着重分析了优化器的核心部分：执行路径的生成与选择。指出了其选择算法存在的缺陷，提出了一个改进的算法。

关键词　请求优化，请求优化器，PostgreSQL，路径，规划，基因优化

Research on the Mechanism of Query Optimization in the PostgreSQL

LIU Chang-Hao　SUN Yu-Fang
(Institute of Software, Chinese Academy of Sciences, Beijing 100080)

Abstract　As the most powerful and complex free DBMS with the most features, PostgreSQL not only draws the DBMS researchers' & developers' attention, but also is adopted in commerce more and more. It seems that it has a very promising future. This article discusses the implementation of the Query Optimization of the PostgreSQL, and analyzes the kernel part of the optimizer, the generating and choosing of the execution path. And, it points out a shortage of the optimization arithmetic, give a mended arithmetic.

Keywords　Query optimization, Query optimizer, PostgreSQL, Path, Plan, GEQO

1　简介

在一个数据库管理系统中，对于一个用户请求，数据库管理系统(DBMS)有很多不同的规划(plan)来执行这个请求。这些不同的规划给终端用户(end user)返回相同的结果，但是却具有不同的效率。这些不同的规划由规划器(Planner)生成，而优化器(Optimizer)则负责根据某些规则(比如代价模型(Cost Model))，从这些不同的规划中选择执行时代价最小的规划，然后把这个规划交给执行器(Executer)去执行。这样一个过程在数据库管理系统中被称为请求优化(Query Optimization)。事实上，通常人们把上面所说的规划器和优化器统称为请求优化器(Query Optimizer)。请求优化对于提高关系数据库的性能，特别是对于复杂的 SQL 请求，有着非常重要的作用。它在请求处理流程中的所处的位置如图1所示。

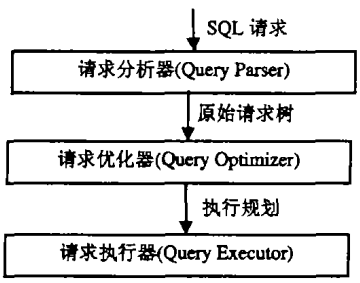

图1　请求处理流程(Query Processing)

PostgreSQL 是一种非常复杂的对象-关系型数据库管理系统(ORDBMS)，也是目前功能最强大、特性最丰富和最复杂的自由软件数据库系统。它的很多特性正是当今许多商业数据库的前身，而有些特性甚至连商业数据库都不具备[2]。它起源于伯克利(BSD)的 Ingres 项目，具有以下一些鲜明特点：

1)支持用户自定义数据类型(UDT)、操作符和函数(UDF)，这为描述复杂和独特的数据提供了手段。有些数据比较特殊，只用于比较特别的行业，因此通用的数据库管理系统不可能也不应该把这样的数据类型包含到它的发布版本中。但是，对于某个行业，这样的数据类型又是必需的，用户自定义数据类型(UDT)和操作符(UDF)就为满足这样的需求提供了可能。

2)支持复杂对象数据类型(Complex Objects)的构建。许多领域的数据，不同于一般的商业数据，它们通常都是许多维数据的集合体，如果用通常的关系数据库管理系统(RDBMS)来存储，则每维的数据都必须用一个表来模拟；这样对一个对象的有关操作，就会涉及到多个表的访问，从而影响效率。数据库管理系统的对象支持技术将描述一类对象所有维的数据构造成一个对象类型，用这个对象作为属性的数据类型存储在表中，这就减少了表间的相互操作从而提高效率。

3)它把请求分为一般的命令请求和复杂的数据查询更新请求。然后采取不同的处理策略，对于一般的命令请求，直接交给命令执行模块进行执行。而对于比较复杂的数据请求，则将其请求传送到重写器(Rewriter)和优化进行优化，然后将优化后的结果交给执行器执行。

另外，PostgreSQL 还支持许多商用数据库拥有的特性，比如支持事务、子查询、多版本并行控制系统、数据完整性检查。但是，PostgreSQL 之所以能够成为最为流行、拥有非常众多的用户的开源数据库[2]，与它的查询优化系统所具有的良好性能有着非常密切的关系。因为对于生成同一个结果的请求的不同的执行路径来说，其效率有可能有天壤之别，有的执行路径的执行效率甚至相差千、万倍。本文将讨论 PostgreSQL 请求优化的核心部分：执行路径的创建及选择，讨论

刘长浩　硕士研究生，研究方向为数据库管理系统与 XML。孙玉芳　教授，博导，研究方向为系统软件、中文信息处理。

其存在的缺点，并将提出一个解决方案。并简单描述 PostgreSQL 的请求优化比较独立的几个部分：代价模型在 PostgreSQL 中的实现——系统表 pg_statistic、平面化子查询(flattening subquery)、基因查询优化(GEQO)。

2 路径(Path)

在 PostgreSQL 的规划/优化过程中，建立不同的路径(path)来表达处理一个请求的不同的方案。选择那个生成目标关系的代价最小的路径，把它转化为一个规划，这个规划被传递给执行器执行。因此优化器的核心就是建立许许多多的路径，然后从中找出最优的那一条。不同的路径主要是由于关系表的不同的访问方式(如顺序访问(Sequential Access)、索引访问(Index Access))和关系间不同的链接方法(嵌套循环链接(Nest-loop Join)、归并链接(Merge Join)、哈希链接(Hash Join))、链接顺序(左链接(Left-join)、右链接(Right-join)、布希链接(Bushy-join))造成的。在优化器中，使用系统表 pg_statistic 中的系统统计信息来估计不同的路径的代价(Cost)。

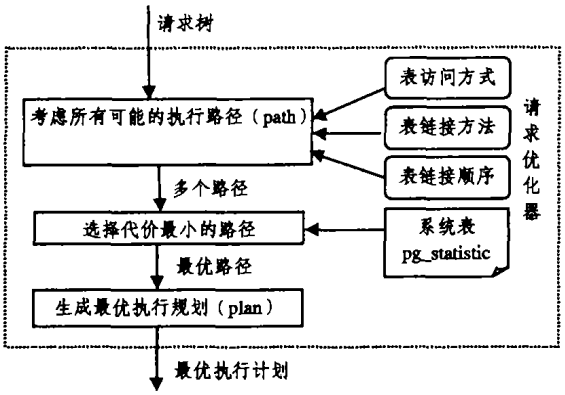

图2 请求优化器体系结构图

在规划阶段，使用相同的数据结构 RelOptInfo 描述一个链接表(Join Relation)以及参加链接的基表(Base Relation)。优化器为在请求中的每一个基表建立了一个 RelOptInfo 的数据结构。基表或者是原始表(Primitive Table)，或者是通过一个独立的规划器的递归调用所规划出的一个子查询(Subquery)的结果。同时，对于每一个在规划阶段所涉及的连接表，都要为它建立一个 RelOptInfo 数据结构。一个连接关系就是若干个基表在一定的约束条件下的简单的合并。对于任何给定基表的集合，只有一个链接的 RelOptInfo 结构。比如：链接{a,b,c}是用同一个 RelOptInfo 来表示，无论对于 a,b,c 这三个基表的连接的顺序是如何。RelOptInfo 的域 relids 是一个链表结构，用来标识这个数据结构描述的是一个基表还是一个链接表。如果这个链表中只有一个元素，这个 RelOptInfo 数据结构表示的就是一个基表；否则就是一个链接表。

对基表的不同的访问方法、对连接表的不同的链接方法、不同链接顺序的不同组合，就构成了一个个不同的路径，以链表的形式存储在 RelOptInfo 的 pathlist 域中。链表 pathlist 的每一个成员都是一个 path 节点。

对于一个原始的关系(Primitive Relation)，可能的路径包括简单的顺序访问，以及表上存在的任何的索引的索引访问。而一个链接表的一个路径实际上就是一个树结构(Tree)，其最顶端的节点表示了链接的方法。它有左右两个子路径(Subpath)，分别代表输入的两个表所使用的扫描(输入为基表)或连接(输入为链接表)的方法。链接通常涉及到两个 RelOptInfo，对应于参加链接的两个表。一个是外部表(Outer Relation)，一个是内部表(Inner Relation)。外部表的 RelOptInfos 驱动内部表值的查找。在嵌套循环的链接中，内部表值的查找是通过对每一个外部表的元组(Tuple)扫描整个内部表来发现每一个与之相匹配的内部行。在归并连接里，内部表和外部表的行都是需要预先排序了的，按顺序访问，因此只需要一次扫描就可以完成整个连接：内部表和外部表的路径都被同步地扫描(在归并连接中，外部和内部没有太大的区别。)。在哈希连接中，内部表首先被扫描，所有它的行被放到一个哈希表中，然后，外部表被扫描，对于每一行，在哈希表中查找链接关键字进行链接。

3 路径关键字(Pathkeys)

PathKeys 链表是数据结构 path 的一个域，它描述了一个特定的路径的已知的排序顺序方面的信息。Pathkeys 是一个由 pathkeyitem 节点所连接成的链表们所连成的一个链表。它描述了 path 所产生的结果的排序顺序，第 i 个子链表描述了结果的第 i 个排序键。对于每一个 pathkeyitem，都有一个排序操作符和它相关，因为跨数据类型的归并链接也是有可能；例如 int4=int8。

顶层链表的顺序是有意义的(不同级别的路径关键字)，每个子链之中成员的次序是任意的。每一个子链应该被认为是具有相同地位的关键字的集合，其中的顺序没有任何意义。这样，从每个不同的子链中取出一个变量，这些变量就形成了访问这个基表或是构建这个链接关系的一个特定的路径。

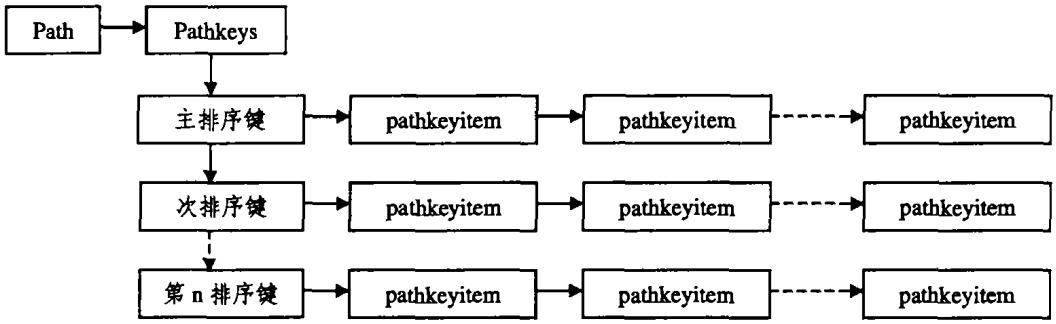

图3 路径关键字(Pathkeys)的结构

在简单的基表的 RelOptInfo 里，路径描述的是扫描表的不同的方法，以及元组排序的不同的结果。顺序扫描的路径的路径关键字是 NIL，表明没有已知的排序。如果有索引扫描的话，索引扫描路径的路径关键字描述了所选择的索引排序。一

个单关键字的索引将会创建只有一个子链表的路径关键字。多个关键字的索引对每个关键字生成一个子链表(sublist),例如((tab1. indexkey1 / sortop1)(tab1. indexkey2 / sortop2)),这表明使用 indexkey1作为主排序,indexkey2作为辅排序。

一个多步完成的索引扫描(最后的结果是多次索引扫描结果的并集)或是子句扫描的路径关键字是 NIL,因为根本不能够预测它的结果的全部顺序。在一个未知类型的索引上的索引扫描也会产生一个 NIL 的路径关键字。然而,总是可以通过一个显式的排序操作来创建一个路径关键字。

当为一个归并(merge)或是嵌套循环链接构造路径关键字的时候,可以保留所有外部路径的关键字,因为外部路径的顺序将会在结果中继续保持。而且,可以把那些在子链接中和任何一个外部变量进行相等链接的那些内部变量交到每个路径关键字的子链表中;无论是如何实现那个链接的,是把相等链接作为一个归并子句或是仅仅强制那个子句作为事后一个基因条件的过滤器,这都将是起作用的。

尽管哈希链接也能够工作在只有相等链接的操作符下,认为哈希链接的输出是以任意特定顺序(即使是以外部路径的顺序)进行排序是不安全的。因为执行器可能不得不把链接分割为多个处理单元。因此,哈希链接的路径关键字总是 NIL。

实际上,路径关键字在描述一个希望获得的排序方面是有作用的,因此排序子句列表在优化器内部被转化为路径关键字的链表。

4 路径关键字的等价集

如果像上面描述的那样实现路径关键字,就会发现规划器花费了大量的时间来比较路径关键字,因为路径关键字作为无序列表的表示方式使它在决定两个路径关键字是否相等时的花费太大。下面引入等价集合(equivalence set)的概念来简化路径关键字的比较,将会发现引入等价集合之后复杂性大大减小。

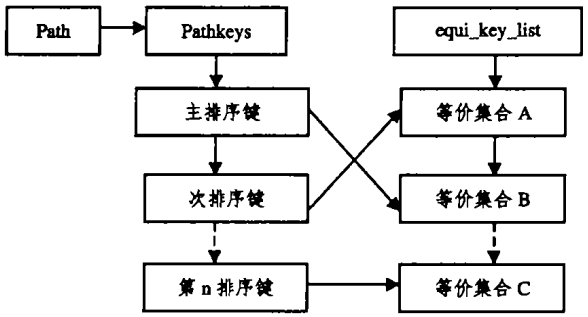

图4 引入等价集合后路径关键字的结构

如果在规划器启动的时候扫描 where 子句以获得相等链接子句(可归并链接子句),可以为请求创建相等 pathkeyitem 节点(属于同一个等价集合)的列表(在 PostgreSQL 中,这个链表就是 equi_key_list)。在每个等价集合里面可能有超过两个的元组,例如 WHERE $A.X = B.Y$ AND $B.Y = C.Z$ AND $D.R = E.S$ 创建了等价集合{$A.X$ $B.Y$ $C.Z$}和{$D.R$ $E.S$}(加上相关的排序操作符)。属于一个等价集合的任何一个 pathkeyitem 隐含着在这个集合中的所有其它元素也应用到这个关系,或者至少所有在这个关系中出现的那些域(在那个集合中的某些元素可能只是为了还没有联接的关系)。而且,在规划请求的任何一个阶段出现的任何一个含有多个元素的 pathkey 子链表必定是一个或另一个这些等价集合的一个子集;除非两个元素在 where 子句是相等,否则它们不可能出现在同一个 pathkeys 的子链表中。

现在,我们可以假定一个路径关键字的链表就是一个直接指向一个相关等价集合的指针,这个集合被保存在由这个请求的 pathkeyitem 等价集合组成的一个链表中(equi_key_list)。这样,子链表 pathkeys 的比较就简化为一个指针是否相等的检查。这样,路径关键字的结构就可以用图4表示。

注意,必须把只含有一个元素的 pathkeys 的子链表加到一个请求的等价集合的链表中。

实际上,等价集合也是由 pathkeyitem 节点组成的子链表。

通过引入等价集合这个概念,还可以对排序相关的其他操作有所帮助。例如:如果有一个 where 子句 $A.X=A.Y$,使用一个建立在 X 上的索引扫描 A,我们可以合理地推断这个路径也是以 Y 排序的;如果 Y 是在其他的连接子句或是 Order by 使用的一个变量,这可能就会有所帮助。因此,任何一个具有可归并操作符的 where 子句都可以产生一个等价集合,即使它不是一个链接子句。

5 链接树(Join Tree)构造

PostgreSQL 是使用动态规划来获得最优的路径树的。其具体的步骤如下:

1)对于请求中的每一个基表,为它建立一个 RelOptInfo 结构。找出访问基表的可能有用的方法,包括顺序的和索引扫描,为每一种方法创建一个路径。为一个给定关系所创建的所有的 path 都被放在它的 RelOptInfo. pathlist(实际上,在进入 pathlist 之前,我们丢弃了那些明显不具有优势的 path,因为最终需要的是在 pathlist 中每个可能有用的关系排序的花费代价最小的方法。)。然后,再建立节点:RelOptInfo. joininfo,这个节点列出了所有的连接子句以及所涉及的关系。例如:where 子句 'tab1. col1 = tab2. col1' 为表 tab1生成了一个 JoinInfo 结构,这个结构把 tab2作为一个没有联接的关系列出来。同样,它也为表 tab2生成了一个 JoinInfo 结构,这个结构把 tab1作为一个没有联接的关系列出来。

2)如果请求的 From 子句包含显式的 join 子句,就按照 join 子句中指定的树的结构来链接这些关系。对于每一个这样的链接对,为每一个可行的连接方法生成一个路径,选择花费最小的路径)。注意,链接子句的结构决定了链接路径的结构,但是它并不约束链接的实现方法,它也没有说明在链接的时候哪个表是外部的,哪个表是内部的。在创建 path 的过程中考虑所有的这些可能性。

3)在 from 子句的顶层,将会有一系列的关系表,这些关系表或者是基表或者是每一个 join 链接符所创建的链接表,能够以规划器看来合适的顺序把这些表连接到一起。标准的规划器(非基因优化的)以动态规划来实现:

step 1. 这一系列单个关系的路径根据上面所描述的路径关键字的等价集的概念来判断其是否相等(即判断它们是否指向同一个等价集合)。这些路径从而被划分为若干个等价类。然后引用 pg_statistic 中的数据估计所有路径的代价,在每个等价类中代价最小的路径被标识以作进一步的考察,而其他的路径就被剪除。但是无序等价类中的最优规划(花费最小)如果比所有其他规划的花费要大,就不会被标识。

step 2. 对于在请求中链接的每一对关系,在 step1后,所

有的访问路径和所有可能用来评估它们链接的方法都已经取得。链接每一个 RelOptInfo 到在它的 RelOptInfo.joininfo 所指定 RelOptInfo，然后对每一对如此的 RelOptInfo 为每一种可能的连接方法生成一个路径(如果一个 RelOptInfo 没有连接子句，连接任何可能的关系。但是如果出现连接子句，就只考虑在链接子句中出现的连接。)如果再 from 列表中仅仅两个关系，那么就为连接 RelOptInfo 选择花费最小的路径就可以了。如果 from 列表中有不止两个关系，就需要考虑进一步链接连接 RelOptInfo 到其他 RelOptInfo 的方法以生成多于两个关系的连接 RelOptInfo。这个部分路径(两个关系的)的划分和剪除就像 step1那样处理。

……

step i. 对于请求中 $i-1$个关系链接的集合，从上一步已经知道由这个集合链接而成的链接关系的代价最小的路径。这一步，对于每一个这样的集合，评估所有可能的使用非笛卡尔的链接更多关系的方法，对于每一个有 i 个关系的集合，所有生成的(部分)路径被上面所描述的那样进行划分和剪除。

……

step n. 所有能够完成这个请求的可能方法从上一步中的规划中获得。花费最小的路径就是优化器最终的输出，它被用来处理这个请求。

在生成部分路径时，它常常通过动态地剪除路径空间次优的路径而避免了列出所有可能的路径。在这个算法中，就是通过剪除每个路径等价类中的非最优路径来实现的。

6 链接树构造算法的改进

上面的链接树构造算法考虑了所有可能的链接树，包括左链接树(上一层链接的外部关系是一个链接表，但是内部关系总是一个 from 的元素)；右链接树(外部关系总是一个简单的元素)；布希链接(Bushy-join)树(内外部关系都可以是链接表)。例如，当构造{1 2 3 4 }时，可以考虑链接{1 2 3}到{4}(左链接)，或者{4}到{1 2 3}(右链接)，或者{1 2}到{3 4}(布希连接)。这样的动态规划算法在最坏的情况下可能具有 $N!$ 的复杂度(4)，这对于计算机的处理是非常大的隐患。实际上，我们可以把这个算法的复杂度降低到2的 N 次方而几乎不影响最优路径的选择(3、4)。这是通过对内部关系的限制来实现，要求：每一个链接的内部关系都是一个原始数据库关系，而不可以是一个中间结果。即只在左链接树空间来搜索最优树，基于以下的两个理由，最佳的左链接树的代价不会比最优树高出太多(3)：

1)使用原始的数据库关系作为内部关系增加了已经存在的索引的使用率。

2)使用中间结果表作为外部关系可以允许嵌套循环链接以通道的方式进行。减少了对中间结果的存储操作。

改进后的算法：(略)。

事实上，改进后的算法尽管总的来说仍然是指数型的，但是它仅仅生成 $O(N^3)$个规划(3)。

7 系统表 pg_statistic

在 PostgreSQL 中，系统表 pg_statistic 担当着代价模型的作用。它存储有关该数据库内容的统计数据，以供优化器使用。记录是由 ANALYZE 和 VACUUM ANALYZE 命令更新的，并且总是近似值，即使刚刚更新完也不例外，随后被查询规划器使用，以生成最优的执行路径。因为不同类型的统计信息适用于不同类型的数据，pg_statistic 被设计成不太在意自己存储的是什么类型的统计。只有极为常用的统计信息(比如 NULL)才在 pg_statistic 里给予专用的字段。其它所有东西都存储在“槽位(slot)”中，而槽位是一组相关的字段，它们的内容用槽位中的一个字段的代码号码表示。当前，定义了三种类型的统计槽位(slot)：大多数普通的值，柱状图(histogram)，相关性。其他的类型将来可能会出现处理非标量(non-scalar)的数据类型。关于这些统计信息的收集，需要考察许许多多的因素，比如 I/O，CPU，每一个请求操作所需要的内存资源，数据库对象(表、索引等等)的统计信息(在 PostgreSQL 中，这需要访问系统表 pg_class 的 reltuples 和 relpages 域)。

8 平面化子查询

一个出现在关系引用列表(From-list)中的子查询被独立规划，在规划外部查询的过程中被当作一个黑盒子。当子查询使用诸如聚集函数(Aggregate)，分组，DISTINCT 属性的时候，这是必要的[5]。但是，如果子查询仅仅只是一个简单的扫描或是链接，相比较把它当作整个规划搜索空间里的一部分，把子查询当作是一个黑盒子可能会产生一个差的规划(一个典型的例子如下)：

平面化以前的 SQL 语句：

```
SELECT D.DNAME FROM DEPT D WHERE D.DEPTNO IN
(SELECT E.DEPTNO FROM EMP E WHERE E.SAL = 10000)
```

对于上面的这个子查询，如果它被独立地规划，那么对于表 DEPT 中的每一个 DEPTNO 值，都要搜索整个表 EMP 以获得相匹配的元组，可以看出，这样的代价是非常大的。

平面化以后的 SQL 语句：

```
SELECT D.DNAME FROM DEPT D,EMP E
WHERE D.DEPTNO = E.DEPTNO and E.SAL = 10000
```

平面化了以后，子查询的部分被合并到主查询中，可以进行统一的优化。在这个例子中，只需要先做一下过滤(E.SAL = 10000)，然后把结果和 DEPT 做一下链接就可以了。大大提高了效率。

一般地来说，在规划处理的开始阶段，规划查找简单的子查询，把它们放到整个主查询的连接树中。把子查询合并到主查询中可能导致 From-list 连接出现在链接树顶层的下面，每一个 From-list 都被上面所描述的动态规划搜索方法来进行规划。

如果把子查询合并到主查询中导致了一个 From-list 作为另一个 From-list 的直接孩子(在它们之间没有显式的 join 标识)，那么可以把这两个 From-list 合并到一起。一旦这个完成了，子查询就变成了一个外部请求主体的一部分，它就不会约束链接树的搜索空间了。然而，这可能导致规划时间的难以忍受的增长，因为动态规划搜索的运行时间的复杂度是所涉及 From-list 中元组数目的指数增长的。因此，如果结果会导致在一个链表中含有太多的元组，就不进行 From-list 的合并。

9 基因查询优化(GEQO)

通过上面的介绍，可以知道在 PostgreSQL 中，优化器所

采用的是动态规划优化算法这样一个近似穷举搜索(near-exhaustive search)的策略。它生成一个近乎最优的连接顺序，但是可选路径的数目是随着参加链接的表的个数呈指数倍增长的。这样就使得普通的 PostgreSQL 查询优化器不适合处理非常复杂的(参加链接的表数目大于10)的请求。德国弗来堡矿业及技术大学自动控制系的成员在试图把 PostgreSQL DBMS 作为用于一个电力网维护中做决策支持的知识库系统的后端时，碰到了上面的问题。他们提出了一种新的优化策略，这就是基因查询优化。

基因查询优化是基于一种启发式的优化方法 (heuristic optimization method)——基因算法(GA)的，它通过既定的随机搜索进行操作。具体的基因算法请参照文[6]。

在 PostgreSQL 里的 GEQO 实现的一些特性是[2]：

1)使用稳定状态的 GA(替换全体中最小健康度的个体，而不是整代的替换)允许向改进了的查询规划快速逼近。这一点对在合理时间内处理查询是非常重要的。

2)边缘重组交叉(edge recombination crossover)的使用特别适于在用 GA 解决 TSP(旅行推销员问题)问题时保持边缘损失最低。

3)否决了把突变(Mutation)作为基因操作符的做法，这样生成合法的 TSP 漫游时不需要修复机制。

GEQO 模块让 PostgreSQL 查询优化器可以通过非穷举搜索有效地支持大的连接查询。基因查询优化算法采用一种不同于传统优化算法的策略，有着自己不同的特点，将在下一篇论文里作详细的论述。

总结 影响传统查询优化效率的因素主要有以下一些：

1)一个请求中的选择、投影、链接的执行顺序的选择。

2)访问某个关系时访问方法的选择：顺序扫描、索引扫描。

3)链接方法的选择：嵌套循环链接、归并链接、哈希链接。

4)链接顺序：左链接、右链接、布希链接。

PostgreSQL 使用动态规划算法，考虑了这些因素几乎所有可能的组合(路径)，并通过 pg_statistic 中的统计数据来评估每个路径的代价，以选出最优的。

PostgreSQL 中的优化器还实现了其他传统查询优化技术，比如平面化子查询。实现了等价集合的概念，简化了路径的比较。支持基因优化算法，为处理复杂的链接查询提供了可能。本文还提出了对于路径构造算法的一个改进，使其复杂度明显降低。

参 考 文 献

1 PostgreSQL Development Group. PostgreSQL V-7.3.4 source codes. PostgreSQL website http: www.Postgresql.org, 2003

2 PostgreSQL Development Group. PostgreSQL V-7.3.4 Documentation. PostgreSQL website http:www.postgresql.org,2003

3 Ioannidis Y E. Query Optimization Computer Sciences Department University of Wisconsin Madison, 1998

4 Chaudhuri S. An Overview of Query Optimization in Relational Systems. surajitc@microsoft.com 1998

5 Stonebraker M. The design and implementation of the POSTGRES query optimizer http: www.postgresql.org,1989

6 Comp.ai.genetic. The Hitch-Hiker's Guide to Evolutionary Computation

(上接第150页)

插件，将插件的相关接口定义为抽象类(在 C++ 中是纯虚接口)。本系统为 Delphi 实现，采用第三种方式。

3 相关研究

“单个”防火墙的缺陷启发研究者从“协同”与“统一”角度寻找解决方法，即本文关注内容：集成。有以下几种解决办法：防火墙带，它松散布置于物理网络不同位置，由一个管理配置中心和一组位于网络不同物理位置的防火墙组成[3]，该方式未涉及如何集成。分布式防火墙1999年最先由 AT&T 实验室的 Steven M. Bellovin 提出[2]，主要解决了传统防火墙安全拓扑依赖性问题，该方案不涉及防火墙的异构问题，其硬件实现方案被称为嵌入式防火墙[6]。入侵检测领域也有“集成”、“协作”主题的研究，包括：NIDES(Next Generation Intrusion Detection System)[7]、AAFID(Autonomous Agent For Intrusion Detection)[8]等，但它们并未研究中心控制端的可插入性研究。在入侵检测和防火墙协作方面，也未见研究。

结论 本论文提出了一个入侵检测和防火墙的集成管理框架，并讨论其结构和框架的组成，给出了部分框架运作的实例，包括：系统注册、通讯协议解析等。最后给出了插件的多种实现方式。程序的开发实践证明，该框架具有灵活的扩展机制，能够在不修改主程序的情况下，轻松扩展各种被集成系统。而防火墙和入侵检测的协作，也很好地阻断了攻击行为。

参 考 文 献

1 Reilly M, Stillman M. Open infrastructure for scalable intrusion detection. In: Proc. of Information Technology Conf. IEEE, Sep. 1998

2 Bellovin S M. Distributed Firewalls. login: magazine, special issue on security, Nov. 1999

3 Stout B. Firewall Farms Whitepaper. www.geocities.com/ResearchTriangle/3372/firewall farms.html

4 Simth R N. Firewall Implement In A Large Network Toplogy. In: Proc. IEEE Future Trends of Distributed Computing Systems, Oct. 1997

5 Snort2.0 Detection Revisited. http://www.sourcefire.com/technology/whitepapers/sf_snort20 detection_rvstd.pdf

6 Prevelakis V. Designing an Embedded Firewall/VPN Gateway. http://www.prevelakis.net /Papers/vpn8.pdf

7 System Design Document: Next-Generation Intrusion Detection Expert System:[Report]. March 1993

8 Balasubramaniyan J S, Garcia-Fernandez J O. An Architecture for Intrusion Detection using Autonomous Agents: [COAST Technical Report]. June 1998